A Toolset for Automated Failure Analysis

Leonardo Mariani[‡]
mariani@disco.unimib.it

Fabrizio Pastore[‡]
pastore@disco.unimib.it

Mauro Pezzè[†,‡]
mauro.pezze@unisi.ch

[†]University of Milano Bicocca
20126, Milan, Italy

[‡]University of Lugano
6904, Lugano, Switzerland

Abstract

Classic fault localization techniques can automatically provide information about the suspicious code blocks that are likely responsible for observed failures. This information is useful, but not sufficient to completely understand the causes of failing executions, which still require further (time-consuming) investigations to be exactly identified.

A useful and comprehensive source of information is frequently given by the set of unexpected events that have been observed during failures. Sequences of unexpected events are usually simple to be interpret, and testers can guess the expected correct sequences of events from the faulty sequences.

In this paper, we present a tool that automatically identifies anomalous events that likely caused failures, filters the possible false positives, and presents the resulting data by building views that show chains of cause-effect relations, i.e., views that show when anomalous events are caused by other anomalous events. The use of the technique to investigate a fault in the Tomcat application server is also presented in the paper.

1 Introduction

In most software projects, large part of the total development effort is spent to localize, identify and fix faults. To reduce this effort, researchers defined several techniques and tools to (semi-)automatically localize faults [13, 10, 7, 14]. Even if many of these techniques can provide useful information in terms of code blocks likely responsible for a given failure, they provide little support to interpret and understand faults, which is almost entirely left to the intuition of developers and testers.

To better understand failure causes, information about suspicious code blocks can be complemented with data about the anomalous events that likely caused a given failure. Sequences of unexpected events are usually simple to be interpreted, and can be easily juxtaposed to the expected sequence of legal events. For instance, if we apply techniques for fault localization to identify the cause of the Apache Tomcat failure documented at `http://issues.apache.org/bugzilla/show bug.cgi?id=40820`, we would identify a few code fragments in `JspFactory` and `JspServlet` classes as likely responsible for the failure. This information is correct and useful, but it does not fully explain the failure. On the other hand, if we look at the anomalous events, i.e., events that are not usually observed in legal executions, we can discover that the `defaultFactory` attribute is exceptionally `null` in the failing execution, and that the invocation of the static constructor of class `JspFactory` has been unexpectedly anticipated. This information can be extremely effective to identify, understand and fix faults.

Techniques that can automatically detect anomalous behaviors work in two main phases: training and verification. In the training phase, they monitor the target system, collect sequences of events generated during legal executions, and synthesize models that summarize and generalize all the events that have been observed. In the verification phase, they compare events observed in failing executions with the learned models, to find the unexpected sequences of events likely responsible for failures. Anomaly detection techniques essentially differ for the type of collected data and the kind of generated models [12, 6, 9, 2, 3].

The major limitation of anomaly detection techniques is the generation of a high number of false positives, i.e., many of the events that are initially indicated as anomalous result to be legal in effect. As a consequence, testers spend a large amount of time in inspecting non-erroneous events, with a huge reduction of the cost-effectiveness of these techniques. Moreover, when a single fault causes multiple anomalous events, e.g., like in the Tomcat example, the anomalous events are not presented to testers with an aggregated view, but they are presented as many single anomalies. Thus, testers are forced to inspect multiple times the same conceptual problems.

In this paper, we present the BCT technique, and the corresponding toolset, that automatically identifies anomalous events, filters likely false positives, aggregates related anomalous events and presents the resulting data with effective views. Anomalous events are identified by initially generating models of the interactions between components, and then comparing these models with events generated during failing executions.

To eliminate false positives, anomalous events detected in failing executions are processed in three steps before being presented to testers. In the first step, we automatically remove the anomalous events that are likely unrelated with the observed failure. In the second step, we use clustering algorithms to automatically generate clusters of likely related anomalous events. In the third step, we eliminate the clusters that resemble the structure of false positives, and we prioritize the remaining clusters. Our solution visually presents the resulting ordered set of clusters to testers. Clusters provide a coherent and complete view of the detected problems. This analysis is completely automated and can provide a significant reduction of debugging effort.

The paper is structured as follows. Section 2 presents the BCT technique. Section 3 presents the related toolset. Finally, Section 4 provides conclusions.

2 Automated Failure Analysis

The BCT technique works in four phases, as shown in Figure 1. In the first phase (Capture Behavior), we execute test cases and capture interactions between components. We record both sequences of inter-component method invocations and data exchanged between components.

In the second phase (Derive Models), we generate models that summarize and generalize the observed executions. In particular, we use Daikon to generate properties about the exchanged data [4] and kBehavior to generate finite state automata (FSA) that represent component interactions [9]. For instance, if we monitor invocations of method `addItem(Item it, int qt)`, we can automatically generate the property `qt>0` that indicates a positive value for the quantity. Similarly, if we monitor the use of a library for accessing files, we can automatically generate an automaton that indicates that files are first open, then read several times and finally closed.

In the third phase (Detect Anomalies), the events generated during a failing execution are checked with the inferred models. Checking can take place either in-the-field, while the system is under use, or in-house, when a failing execution is replicated to debug a failure. Events that violate models, i.e., data values that violate properties and sequence of method invocations that are not accepted by FSA, are collected as anomalous events that deserve further investigations.

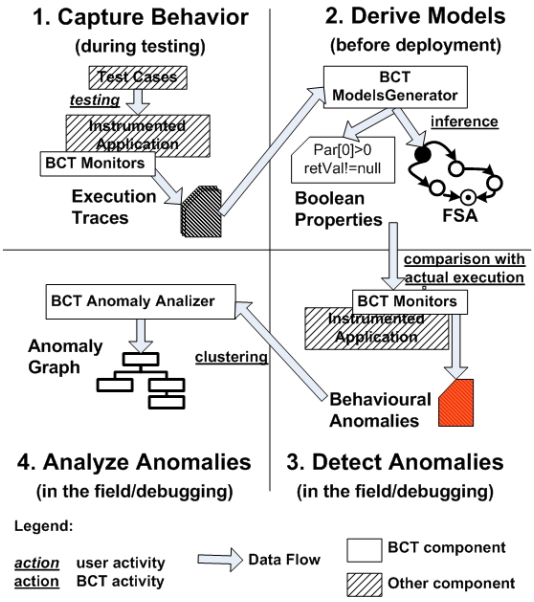

Figure 1. The BCT technique.

In the fourth phase (Analyze Anomalies), the detected anomalous events are filtered, aggregated and presented with suitable views. In the filtering step, we remove the anomalous events that are observed in both successful and failing executions. The rationale is that the events highly related to failures should be observed only when failures occur. On the contrary, anomalies observed independently from failures are likely to be false positives and can be ignored.

The filtered set of events are then aggregated according to likelihood to be related to a same run-time problem. In fact, a single run-time problem can generate many anomalous events. A typical example is the generation of an exception that may cause the violation of many interaction and data models, before being handled by the application. The inspection of each single anomalous event is not a cost effective strategy, and does not provide to testers an adequate and complete view of the run-time problem that affected the system. To build a precise presentation of the detected problems, we first measure distance between anomalous events and then we automatically build clusters of related anomalies. Distance between two anomalous events that have been detected during the execution of methods `m1` and `m2` is measured as the minimum number of nodes that need to be traversed to move from the node that corresponds to `m1` to the node that corresponds to `m2` in the dynamic call tree [1] of the failing execution. The rationale is that related anomalies are frequently observed close to each other. In fact, corrupted executions usually run abnormally and generate many anomalous events until the application recovers

or crashes. The clustering algorithm used to isolate closely related anomalous events is the Within Clustering Dispersion [5].

Each cluster conceptually represents a possible explanation to a different run-time problem that has been observed. To further reduce the number of false positives that can be incidentally presented to testers, we eliminate clusters composed of a single node (outliers), which likely represents false positives incidentally included in the final result.

Finally, clusters are prioritized according to their size. In fact, complex clusters frequently describe complex and highly unexpected executions that the system did not handle correctly, while the simplest clusters are usually less relevant. Results reported in [8] show that the final set of prioritized clusters provide an effective explanation of observed failures by inspecting only few entries.

Figure 2 shows the result generated by our technique for the Tomcat fault presented in Section 1. A high number of false positives have been automatically filtered and the resulting cluster is extremely simple, easy to interpret and effective in explaining the cause of the failure. The only selected cluster includes two nodes: one indicating an unexpected `null` value due to a postponed method invocation (indicated by the second node).

Note that nodes within a cluster are ordered according to order of appearance of the corresponding unexpected events, and each cluster has at a list a root node. Root nodes indicate the first event of the cluster that should be analyzed by testers, while the links indicate a cause-effect relation between anomalies, i.e., anomalous event e_1 is linked to e_2 if e_1 likely caused e_2. Multiple root nodes may indicate either multiple causes of a problem or an imprecision in the construction of the cluster.

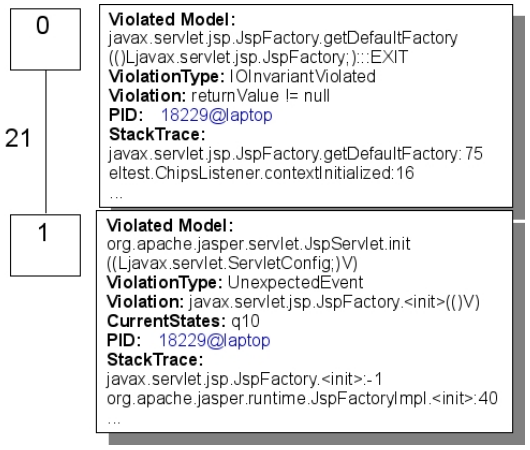

Figure 2. The result for the Tomcat fault.

3 A Toolset for Automated Failure Analysis

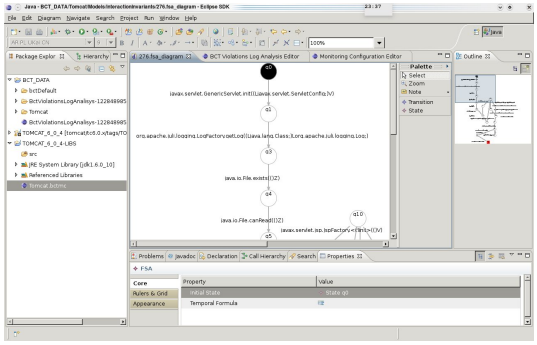

Figure 3. The tool environment.

BCT is implemented as an Eclipse Plugin and can be used to analyze Java programs. Figure 3 shows a screenshot of the application, Figure 4 shows the architecture of the core components of the tool, and Figure 5 shows the architecture of the presentation layer.

In the capture behavior phase, we use either aspects or TPTP probes to record interactions between components. Aspects/probes are automatically generated by our tool from a specification of the components to be monitored that is provided by testers. The recording aspects/probes integrate the object flattener component that can inspect private fields of the objects that are exchanged between components. For instance, if a component exchanges a parameter of type `Person` with another component and the interaction is captured, the object flattener recursively inspects attributes of the `Person` object to extract the concrete values that will be used to generate data models.

In the model generation phase, the Model Generator component generates models from collected data. In particular, the Model Generator works as a driver of the different inference engines integrated in our technology. If necessary, a pre processing component can be used to transform the collected data to be compatible with the specific inference engines that are used. Additional inference engines can be integrated in our solution by adapting the Model Generator.

The presentation layer provides functionalities to visualize and edit all the generated models.

In the Detect Anomalies phase, our tool uses the same configuration provided for the monitoring, to generate aspects/probes for comparing executions with models. Aspects/probes are injected into the target system for forwarding events to the model comparator that matches events with models, and logs the detected anomalies. Anomalies are recorded in the CBE format [11]. Each anomaly includes information about the name of the component and the values that caused the anomaly, the model that detected the

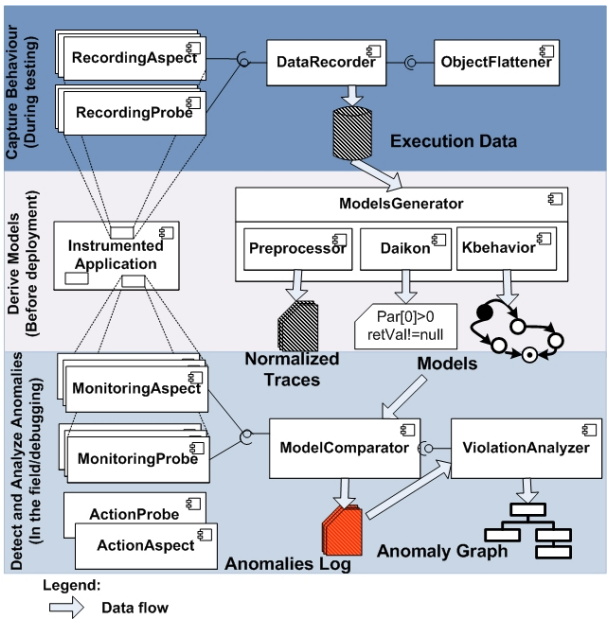

Figure 4. The BCT architecture.

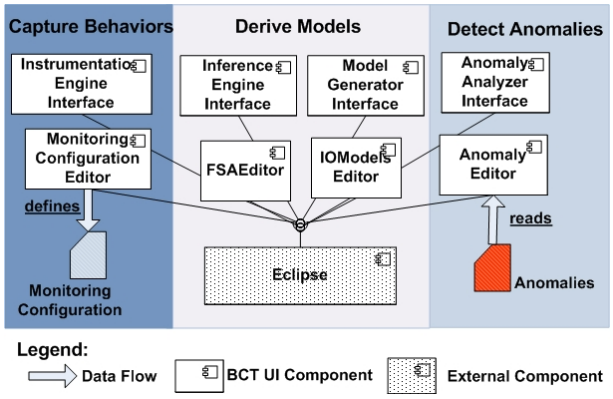

Figure 5. The BCT presentation layer.

anomaly, the stack trace and the process identifier.

Finally, the Violation Analyzer is used to filter, aggregate and present the anomalous events that have been detected. All these steps can be controlled by the presentation layer that provides functionalities to visualize the filtered anomalies and inspect the resulting clusters.

4 Conclusions

To reduce the effort required for fault localization and identification, tools can be used to automate many of the analysis steps otherwise manually executed by testers. In this paper, we presented a technique and a toolset to automatically identify anomalous events in failing executions, filter false positives, aggregate related anomalies and build effective views presenting the result of the analysis. A case study based on a known Tomcat fault is used to show effectiveness of the technology.

Acknowledgment This work has been supported by the European Community under the Information Society Technologies (IST) programme of the 6th FP for RTD - project SHADOWS contract IST-035157.

References

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *proceedings of the Conference on Programming Language Design and Implementation*. ACM, 1997.

[2] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *proceedings of the 29th Symposium on Principles of Programming Languages*, pages 4–16. ACM Press, 2002.

[3] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for java. In *proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP'05)*, 2005.

[4] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001.

[5] A. Gordon. *Classification*. Chapman and Hall/CRC, 2 edition, 1999.

[6] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *proceedings of the 24th International Conference on Software Engineering*, 2002.

[7] J. Jones, M. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *proceedings of the Internation Conference on Software Engineering*, 2002.

[8] L. Mariani, F. Pastore, and M. Pezzè. Dynamic analysis for integration faults localization. Lta internal report, University of Milano Bicocca, 2008. available at www.lta.disco.unimib.it.

[9] L. Mariani and M. Pezzè. Dynamic detection of cots components incompatibility. *IEEE Software*, 24(5):76–85, 2007.

[10] W. Masri, A. Podgurski, and D. Leon. An empirical study of test case filtering techniques based on exercising information flows. *IEEE Transactions on Software Engineering*, 33(7):454–477, July 2007.

[11] D. Ogle, H. Kreger, A. Salahshour, B. Horn, J. Cornpropst, J. Gerken, E. Labadie, M. Chessell, T. Serviceability, J. Schoech, and M. Wamboldt. Canonical situation data format: The common base event v1.0.1, 2004.

[12] O. Raz, P. Koopman, and M. Shaw. Semantic anomaly detection in online data sources. In *proceedings of the 24th International Conference on Software Engineering*, 2002.

[13] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *proceedings of the Internation Conference on Automated Software Engineering*, 2003.

[14] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufman, 2005.